Глава 8. Списки

8.1. Список является последовательностью

Подобно строке, список - последовательность значений. В строке значениями являются символы, в списке - они могут быть любого типа. Значения в списке называются *элементами* (elements) или иногда *записями* (items).
Есть несколько путей создания нового списка, самый простой - заключить элементы в квадратные скобки [ ]:

```
>>> [10, 20, 30, 40]
[10, 20, 30, 40]
>>> ['crunchy frog', 'ram bladder', 'lark vomit']
['crunchy frog', 'ram bladder', 'lark vomit']
```

В первом примере - список из четырех целых чисел, во втором - список из трех строк.
Элементы списка могут быть разного типа. Следующий список содержит строку, число с плавающей точкой, целое число и (внимание!) другой список:

```
>>> ['spam', 2.0, 5, [10, 20]]
['spam', 2.0, 5, [10, 20]]
```

Список внутри другого списка является *вложенным* (nested).
Список, который не содержит элементом, называется *пустым* (empty list), его можно создать с помощью пустых квадратных скобок [ ].

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [17, 123]
>>> empty = []
>>> print cheeses, numbers, empty
['Cheddar', 'Edam', 'Gouda'] [17, 123] []
```

8.2. Списки изменяемы

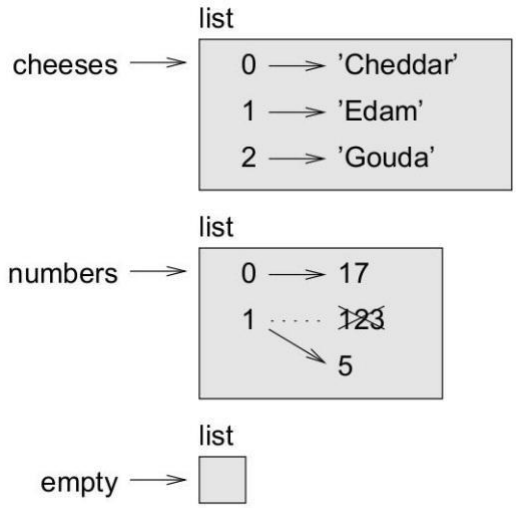
Синтаксис для доступа к элементам списка похож на доступ к символам строки - оператор скобки. Выражение внутри скобок определяют индекс. Запомните, что индекс начинается с нуля:

```
>>> print
cheeses[0] Cheddar
```

В отличие от строк, списки можно изменять, поэтому вы можете изменить порядок элементов в списке или переприсвоить элементы в списке. Когда оператор скобок появляется в левой части выражения, он определяет элемент списка, который будет присвоен.

```
>>> numbers = [17, 123]
>>> numbers[1] = 5
>>> print numbers
[17, 5]
```

Вы можете представить список как связь между индексами и элементами. Эта связь называется *отображением* (mapping), каждый индекс отображается на один из элементов.
Следующая диаграмма состояний показывает списки *cheeses*, *numbers* и *empty*:



Списки на рисунке представлены прямоугольниками со словом список (list) снаружи и элементами списка внутри.

Список индексов работает так же, как и строковые индексы.

- Любое целочисленное выражение можно использовать в качестве индекса.
- Если вы попытаетесь прочитать или записать элемент, которого не существует, вы получите *IndexError*.

- Если индекс имеет отрицательное значение, он ведет счет в обратном направлении от конца списка.

Оператор *in* работает аналогично для списков.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses

>>> 'Brie' in
cheeses False
```

8.3. Обход списка

Наиболее общий путь обхода элементов списка - использование цикла *for*. Похожий синтаксис используется для обхода строк:

```
>>> for cheese in cheeses:
     print cheese


Cheddar
Edam
Gouda
```

Это замечательно работает, если вам необходимо прочитать элементы списка. Но если вы хотите записать или обновить элементы, понадобятся индексы. Общий подход, чтобы это сделать - объединить функции *range* и *len*:

```
>>> for i in range(len(numbers)):
     numbers[i] = numbers[i] * 2

>>> print numbers
[34, 10]
```

Этот цикл обходит список и обновляет каждый элемент. *len* возвращает число элементов в списке. *range* возвращает список индексов от 0 до *n*-1, где *n* - длина списка. За каждый шаг в цикле переменной *i* присваивается индекс следующего элемента. Выражение присваивания в теле цикла использует *i* для чтения старого значения элемента и записи нового значения.

При обходе пустого списка в цикле *for* никогда не выполнится тело цикла:

```
>>> for x in empty:
     print 'This never happens.'
```

Хотя список может содержать другой список, вложенный список считается одним элементом. Длина списка будет равна четырем:

```
>>> len(['spam', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]])
4
```

8.4. Операторы списка

Оператор + объединяет списки:

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

Аналогичным образом, оператор * повторяет список заданное число раз:

```
>>> [0]*4
[0, 0, 0, 0]
>>> [1,2,3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

8.5. Срез списка

Оператор среза также работает для списков:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Если опустить первый индекс, то срез начнется с начала. Если опустить второй - срез закончится в конце. Если опустить оба значения, то срез будет являться копией всего списка.

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Т.к. списки можно изменять то, часто создается копия перед выполнением операций, которые складывают, удлиняют или искажают списки.

Оператор среза в левой части выражения может обновить несколько элементов:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x','y']
>>> print t
['a', 'x', 'y', 'd', 'e', 'f']
```

8.6. Методы списков

Python предоставляет методы, которые оперируют со списками. Например, *append* добавляет новый элемент в конец списка:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print t
['a', 'b', 'c', 'd']
>>>
```

extend принимает в качестве аргумента список и добавляет все элементы:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print t1
['a', 'b', 'c', 'd', 'e']
```

sort организует элементы списка от низших к высшим:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print t
['a', 'b', 'c', 'd', 'e']
```

Большинство методов списка не имеют типа (void), они изменяют список и возвращают *None*. Если вы случайно напишете *t = t.sort()*, то результат вас разочарует.

8.7. Удаление элементов

Существует несколько подходов для удаления элементов из списка. Если известен индекс элемента, можно воспользоваться *pop*:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print t
['a', 'c']
>>> print x
b
>>>
```

pop изменяет список и возвращает элемент, который был удален. Если не указан индекс, то удаляется и возвращается последний элемент списка.

Если вам не нужно удалять значение, можно воспользоваться оператором *del*:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print t
['a', 'c']
```

Если вы знаете элемент, который хотите удалить (но не его индекс), можно воспользоваться *remove*:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

remove возвращает значение *None*.

Для удаления больше, чем одного элемента, можно использовать *del* со срезом индекса:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
```

```
>>> print t
['a', 'f']
```

8.8. Списки и функции

Существует несколько встроенных функций, которые могут быть использованы для списков, они позволяют быстро просмотреть список без написания собственного цикла:

```
>>> nums = [3, 41, 12, 9, 74, 15]
>>> print len(nums)
6
>>> print max(nums)
74
>>> print min(nums)
3
>>> print sum(nums)
154
>>> print sum(nums)/len(nums)
25.666666666666668
```

Функция *sum()* работает только, когда все элементы списка числовые. Другие функции (*max(), len()* и т.д.) работают со списками строк и другими типами, которые могут быть сопоставимы.

Мы можем переписать программу, которая рассчитывала среднее значение чисел в списке с использованием списка.

Сначала программа, подсчитывающая среднее значение, без списка:

```
total = 0
count = 0
while ( True ) :
     inp = raw_input('Enter a number: ')
     if inp == 'done' : break
     value = float(inp)
     total = total + value
     count = count + 1

     average = total / count
print 'Average:', average
```

Можем просто запомнить каждое число, которое вводит пользователь и с использованием встроенной функции подсчитать *sum* и *count* в конце.

```
numlist = list()
while ( True ) :
        inp = raw_input('Enter a number: ')
        if inp == 'done' : break
        value = float(inp)
        numlist.append(value)

average = sum(numlist) / len(numlist)
print 'Average:', average
```

Мы создали пустой список перед началом цикла и затем каждый раз добавляли число в этот список.

8.9. Списки и строки

Строка - последовательность символов, список - последовательность значений, но список символов - это не тоже самое, что строка. Преобразовать строку в список символов можно с помощью *list*:

```
>>> s = 'spam'
>>> t = list(s)
>>> print t
['s', 'p', 'a', 'm']
```

Т.к. *list* - это имя встроенной функции, вы должны избегать его использование в качестве имени переменной. Я избегаю использования *l*, т.к. это похоже на 1, поэтому я использую *t*.

Функция *list* разбивает строку на отдельные буквы. Если вы хотите разбить строку на слова, вы можете использовать метод *split*:

```
>>> s = 'pining for the fjords'
>>> t = s.split()
>>> print t
['pining', 'for', 'the', 'fjords']
>>> print t[2]
the
```

При вызове *split* можно передать аргумент, который задает *разделитель* (delimiter), вместо пробела по умолчанию:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```

join - противоположность *split*, он принимает список строк и объединяет элементы.

```
>>> t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> delimiter.join(t)
'pining for the fjords'
```

В этом случае разделителем является пустая строка, поэтому *join* помещает пробелы между словами. Чтобы соединить строки без пробелов, в качестве разделителя необходимо использовать пустую строку.

8.10. Разбор списков

Обычно, когда мы читаем файл, мы хотим что-то сделать, а не просто вывести на экран всю строку. Часто мы хотим найти "интересные строки", а затем разобрать строки, чтобы найти некоторые интересные части строки. Как быть, если мы хотим распечатать день недели у тех строк, которые начинаются с "From ".

From stephen.marquard@uct.ac.za ***Sat*** *Jan 5 09:14:16 2008*

Метод *split* очень эффективен, когда сталкивается с подобной проблемой. Мы можем написать небольшую программу, которая ищет строки, начинающиеся с "From ", а затем *разделить* (split) эти строки и распечатать третье слово в строке:

```
fhand = open('mbox-short.txt')
for line in fhand:
        line = line.rstrip()
        if not line.startswith('From ') : continue
        words = line.split()
        print words[2]
```

Мы используем условие *if*, которое пропускает все строки, начинающиеся не с 'From '.

Результат работы программы будет следующим:

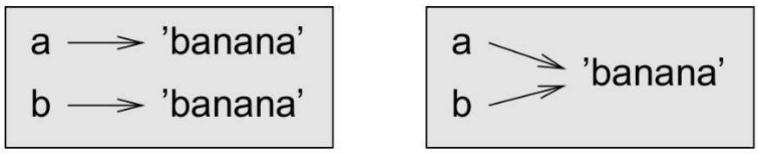
```
Sat
Fri
Fri
Fri
Fri
Fri
...
```

8.11. Объекты и значения

Если мы выполним эти операторы присваивания:

```
a = 'banana'
b = 'banana'
```

Мы знаем, что *a* и *b* ссылаются на строку, но мы не знаем, ссылаются ли они на одну и ту же строку. Возможны два варианта:



В первом случае, *a* и *b* ссылаются на два различных объекта, которые имеют некоторое значение. Во втором случае, они ссылаются на один и тот же объект.

Чтобы проверить, ссылаются ли две переменные на один и тот же объект, вы можете использовать оператор *is*.

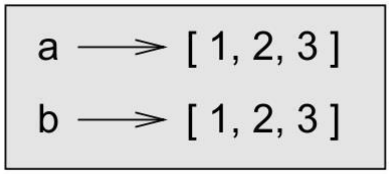
```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True
```

В этом примере Python создает только один строковый объект, *a* и *b* ссылаются на него.

Но когда вы создаете два списка, вы получите два объекта:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

На диаграмме это выглядит следующим образом:



В этом случае мы можем сказать, что два списка *эквивалентны* (equivalent), т.к. имеют одинаковые элементы, но не *идентичны* (identical), т.к. это не один и тот же объект. Если два объекта идентичны, они также эквивалентны, но если они эквивалентны не обязательно, что они идентичны.

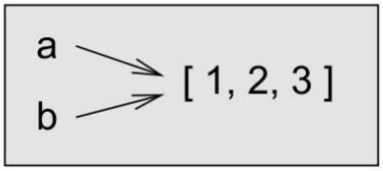
До этого момента, мы использовали понятия 'объект' (object) и 'значение' (value), как синонимы, но более точно говорить, что объект имеет значение. Если вы выполните *a = [1,2,3]*, то переменная *a* ссылается на объект-список, значением которого является определенная последовательность элементов.

8.12. Псевдонимы (Aliasing)

Если *a* ссылается на объект, и вы присваиваете *b = a* то, затем обе переменные ссылаются на один и тот же объект:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

Схема выглядит следующим образом:



Ассоциация переменной с объектом называется *ссылкой* (reference). В этом примере две ссылки на один объект. Объект с более чем одной ссылкой

имеет больше одного имени, поэтому мы говорим, что объект имеет *псевдонимы* (aliased).

Если псевдоним объекта изменяется, то эти изменения касаются других псевдонимов:

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

Хотя такое поведение может быть полезным, существует вероятность ошибиться. В общем, более безопасно избегать псевдонимов при работе с изменяемыми объектами.

Для неизменяемых объектов, таких как строки, псевдонимы не являются большой проблемой.

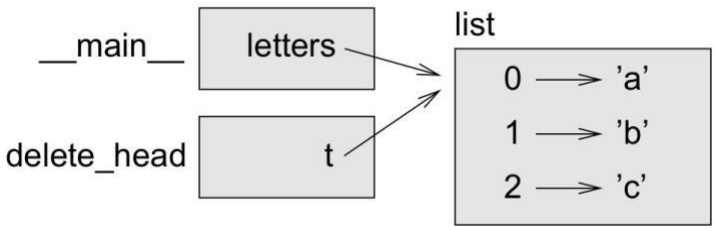
8.13. Список аргументов

Когда вы передаете список в функцию, функция получает ссылку на список. Если функция изменяет параметр-список, вызывающий видит изменения. Например, *delete_head* удаляет первый элемент из списка:

```
>>> def delete_head(t):
     del t[0]


>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

Параметр *t* и переменная *letters* - псевдонимы для одного и того же объекта. Схема стека выглядит следующим образом:



Поскольку список является общим для двух фреймов, я изобразил его между ними.

Важно различать операции, изменяющие списки и операции, которые создают новые списки. Например, метод *append* изменяет список, оператор + создает новый список:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None
>>> t3 = t1 + [3]
>>> print t3
[1, 2, 3, 3]
>>> t2 is t3
False
```

Это различие очень важно, когда вы пишете функции, которые должны изменять списки. Например, эта функция не удаляет первый элемент списка:

```
def bad_delete_head(t):
    t = t[1:] # WRONG!
```

Оператор среза создает новый список и присваивает *t* указатель на него, но это никак не отражается на списке, который был передан в качестве аргумента.

В качестве альтернативы можно написать функцию, которая создает и возвращает новый список. Например, *tail* возвращает все, кроме первого элемента списка:

```
def tail(t):
    return t[1:]
```

Эта функция оставляет первоначальный список без изменений. Вот как она используется:

```
>>> letters = ['a', 'b', 'c']
>>> rest = tail(letters)
>>> print rest
['b', 'c']
```

8.16. Упражнение

1. Приведите примеры списков в Python.
2. Приведите примеры методов списков в Python.
3. Назовите отличие списков от строк.